CAD Tools for Creating 3D Escher Tiles

Mark Howison

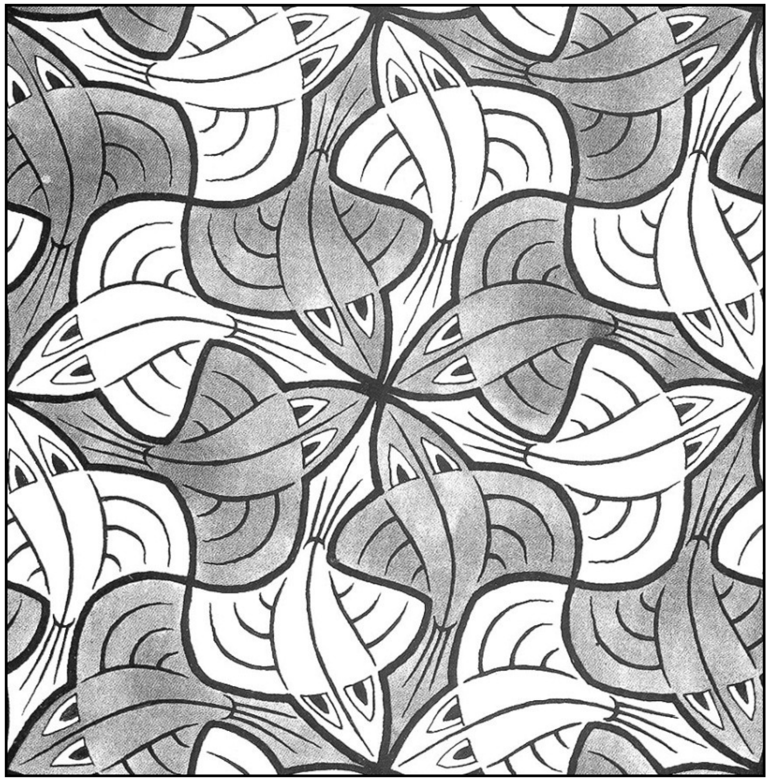
Graphics Lunch, UC Berkeley
February 26, 2009

Overview

- 2½D Tilings
- Incremental Delaunay Triangulation in Java
- Mesh-Cutting Algorithm
- Visual Debugging
- 3D Tilings
- User Interface Issues

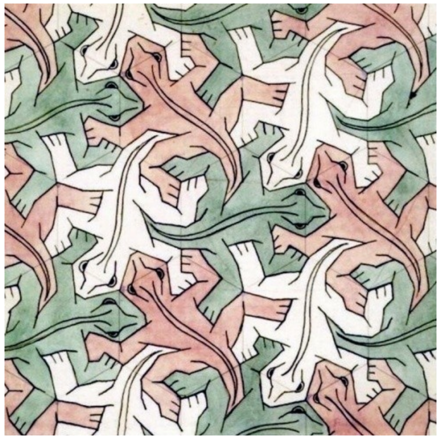
Introduction

- M.C. Escher popularized intricately decorated isohedral tilings
- Planar tilings can be designed by hand, or with available tools on the web
- Specialized CAD tools help address the challenges of tiling other 2-manifolds
- What are interesting tilings of 3-space?



Tiling on 2-manifolds

In the plane



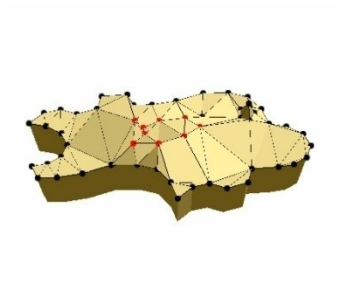
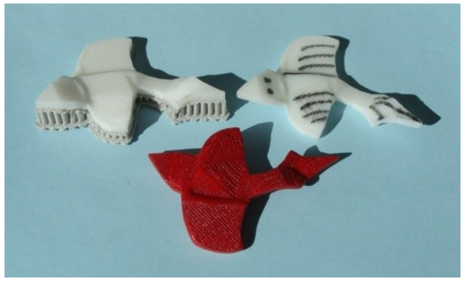
On the sphere

In the
Poincaré disk

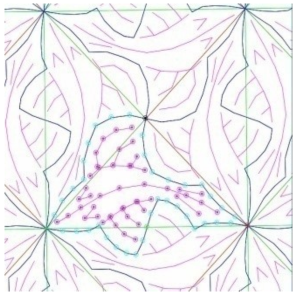
On a genus-3
"Tetrus" surface

2½D Tilings

- Warm-up exercise before tackling full 3D
- Extruded 2D tilings form layers in 3-space
  - Trivial case: extrude vertically, edit height field

  - Fancier case: choose an “offset” between adjacent layers

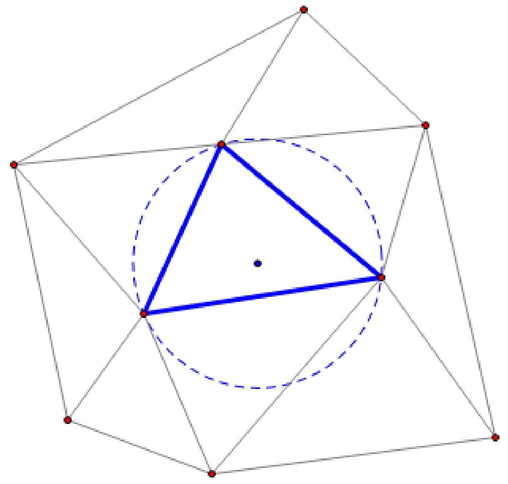


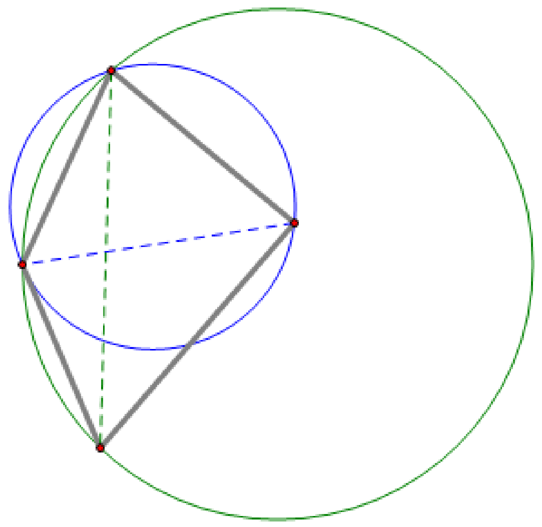
Triangulation Library

- Need meshes for boundary representation of tiles
- Why implement another triangulation library?
  - Chose Java for ease of UI development and portability
    - Could not find existing libraries native to Java
    - Using Triangle would require JNI or dumping/reading ASCII
  - Precise results are unnecessary
    - Limited by precision of Fused Deposition Machine
  - Don’t need optimizations for large meshes, but do need support for frequent modifications
  - Opportunity to learn about data structures and algorithms for triangulation

Delaunay Triangulation

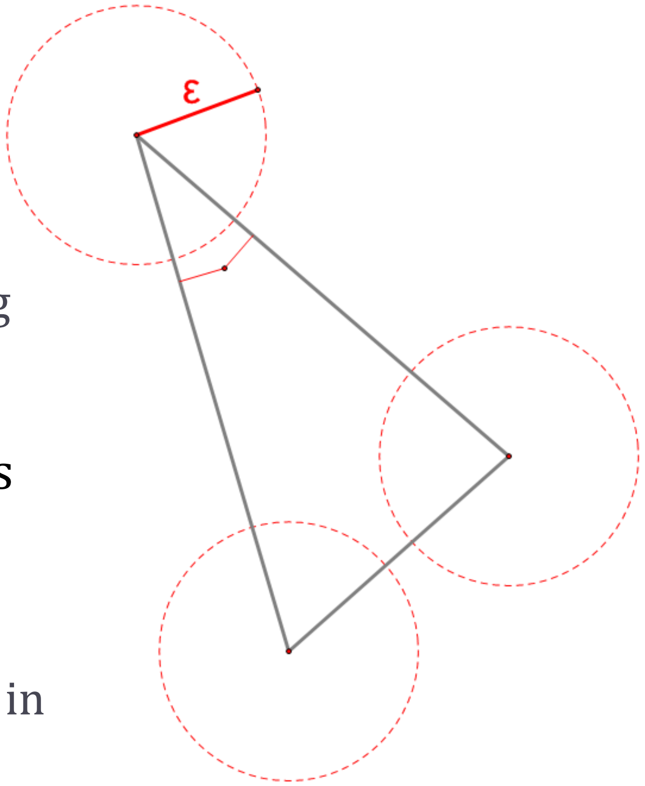
- The circumcircle through any triangle does not contain any other vertices of the triangulation
- "Locally equiangular": maximizes the minimum of the six angles within any two adjacent triangles
- Can use an *in-circle* test to determine if a triangulation is Delaunay
- Can make a triangulation Delaunay by flipping edges





Robustness

- Need reliable tests for determining if a vertex is:
  - Left/right of a line (*orient*)
  - Inside/outside the circumcircle of a face (*in-circle*)
- Exact arithmetic or adaptive precision arithmetic
  - Slow (exact) or complicated implementation (adaptive precision)
  - Precision is not crucial for our application
- Snap to integer coordinates, use integer arithmetic
  - Rotating/skewing/scaling the tile and computing intersections is awkward
  - Modern processors are designed for flops
- Merge vertices that lie within an epsilon radius
  - Use an epsilon >> floating point round-off error
  - Can still have problems with vertices within < epsilon of edges
  - Robust enough for our purposes ("quasi-robust" in Shewchuk's categorization)



Three Basic Types of Algorithms

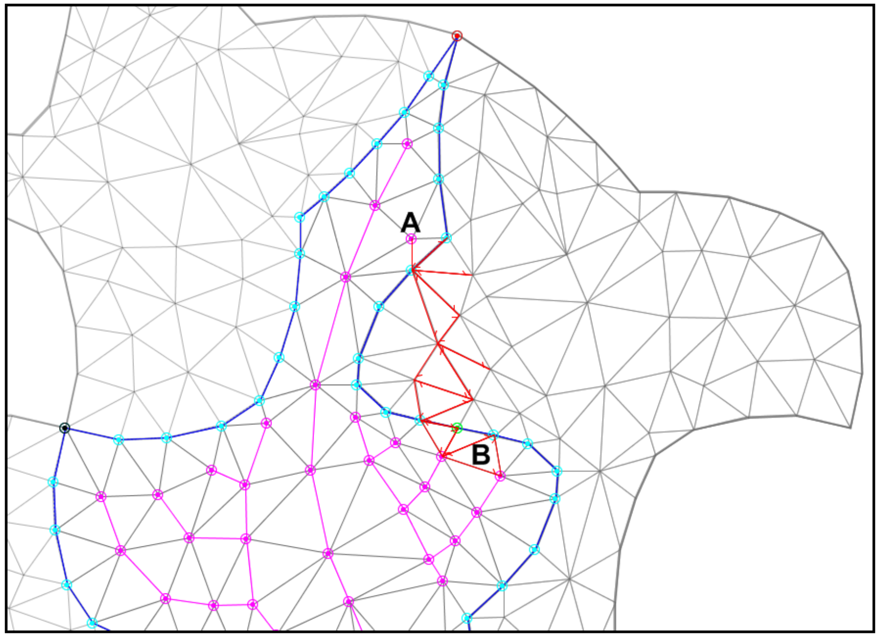
- Divide-and-conquer (e.g., Shamos and Hoey)
  - Typically used to generate the Voronoi diagram
  - Can be modified to return the Delaunay triangulation, the dual of the Voronoi diagram
- Sweepline (e.g., Fortune)
  - Moving “front” along an axis guarantees that circumcircles behind the front can’t contain new vertices
- Incremental insertion (e.g., Lawson)
  - Performance is limited by how well you can locate which triangle contains the insertion site
  - Many optimizations are available

Lawson's Incremental Insertion Algorithm

- Nice properties
  - Maintains the integrity of the mesh after every vertex insertion
  - Easy to implement
- Basic idea
  - Locate triangle containing insertion site
  - Trisect the triangle; add all involved edges to a queue
  - Perform *in-circle* test on queue
    - If an edge fails the test, flip it and add its four neighbor edges back onto the queue
    - Guaranteed to terminate: can show that circumradii are strictly decreasing and there are only finitely many triangulations (Sibson)
- Issues
  - Locating insertion sites is the bottle-neck
  - How do you insert a site outside of the boundary?

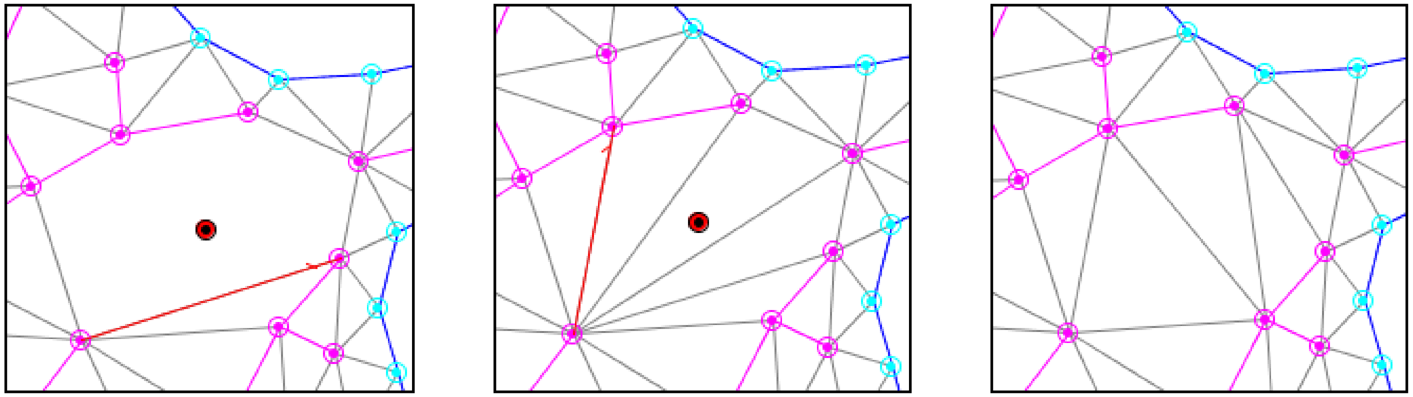
Locating Insertion Sites

- Easy if you have a convex boundary: walk along the triangles
- For non-convex boundaries, we load on-demand copies of the neighboring tiles to fill concavities
- Heuristic:
  use the last inserted site as the search origin, since a designer will often add vertices in localized groups



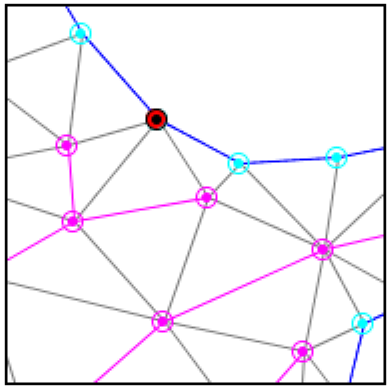
Moving/Removing Interior Vertices

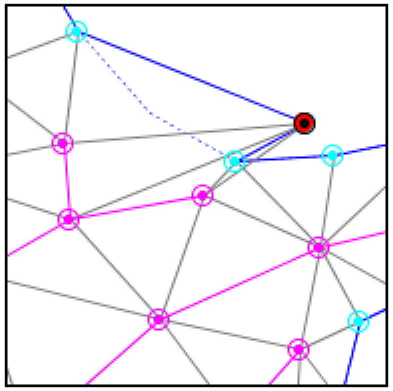
- Temporary holes arise: need polygon filling algorithm
- We use a naïve $O(n^2)$ algorithm for ease of implementation
  - Recursively cuts off "ear" triangles
  - Small polygons, mostly convex: performance isn't an issue
  - If performance were an issue, could use a smarter "trapezoidal decomposition" algorithm

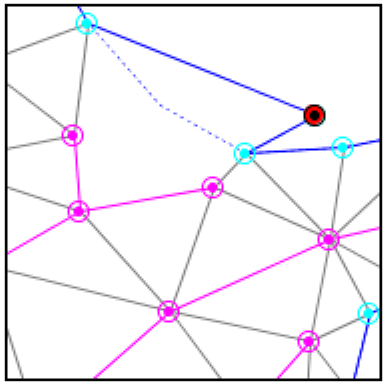


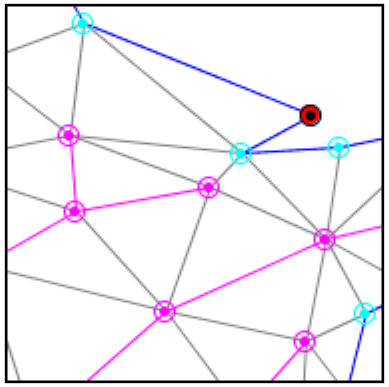
Moving/Removing Boundary Vertices

- Case 1: Move vertex along the boundary
  - Assign new vertex coordinates
- Case 2: Move vertex into the exterior
  1. Assign new vertex coordinates
  2. Remove attached non-boundary edges
  3. Fill polygon left by removal



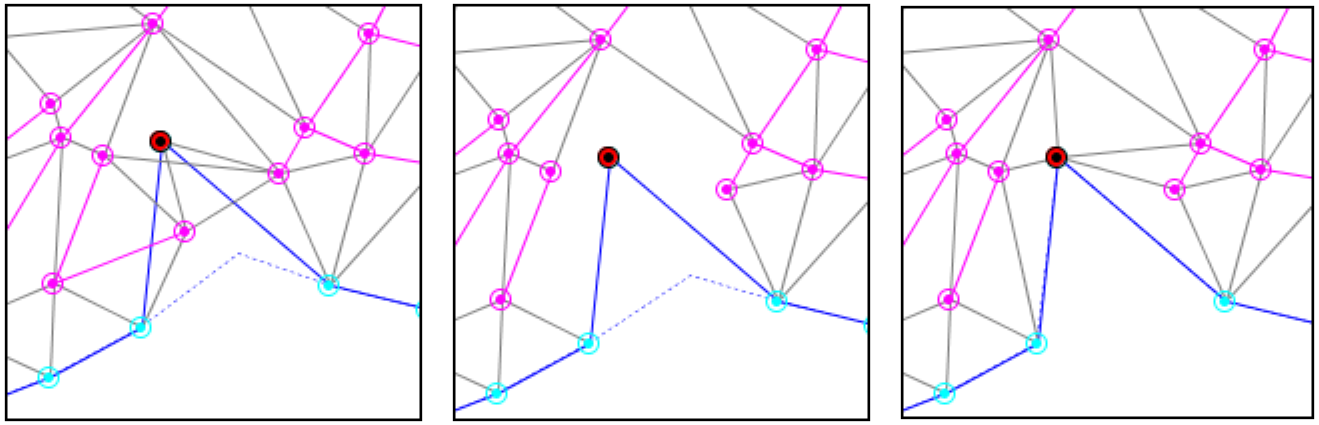

1.


2.


3.

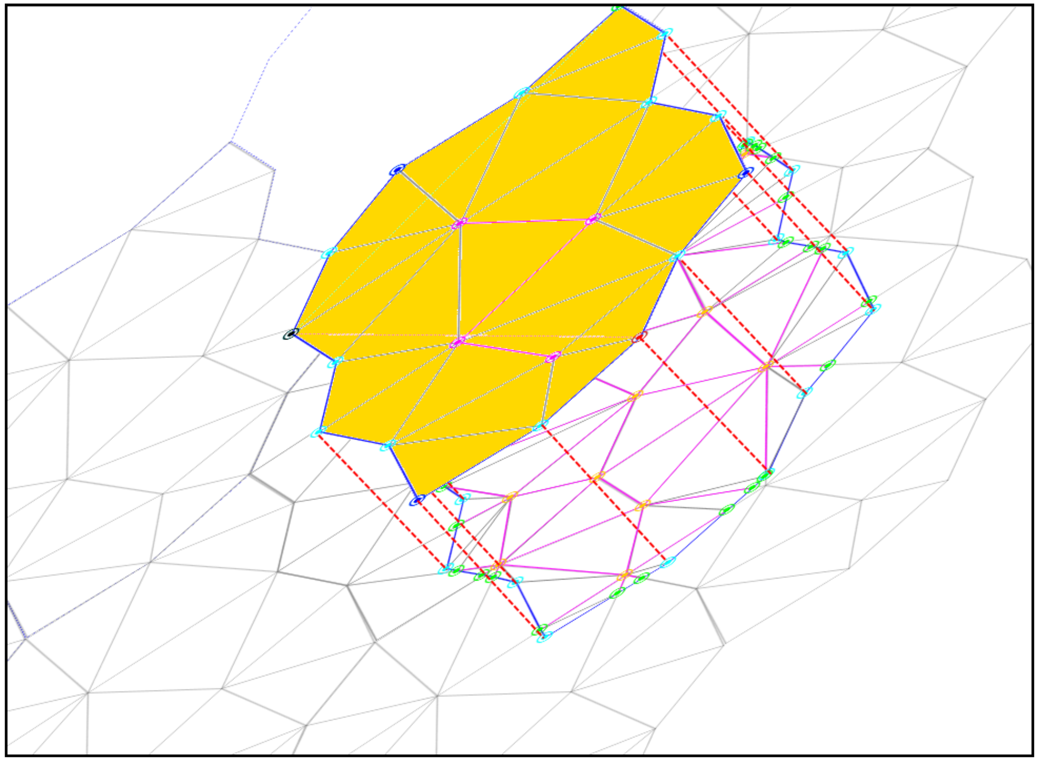
Moving/Removing Boundary Vertices (Cont'd)

- Case 3: Move vertex into the interior
  1. Assign new vertex coordinates
  2. Remove …
     a) attached non-boundary edges
     b) edges that intersect adjacent boundary edges
     c) any exterior edges and points
  3. Fill polygon left by removal



Mesh-Cutting Algorithm

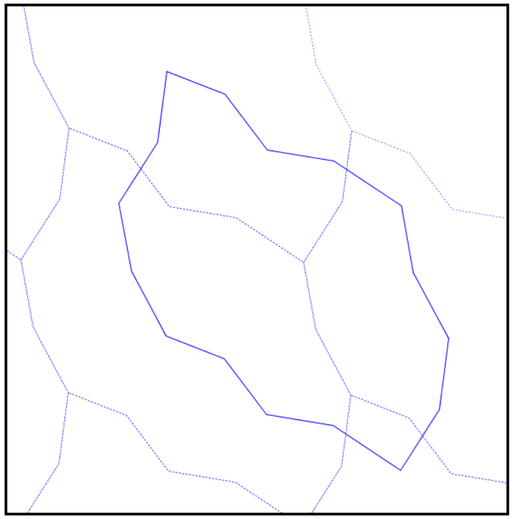
- For forming the bottom face of an offset 2½D tile
- Given an underlying mesh and a "cookie-cutter" template
- Truncates the geometry of the underlying mesh to fit inside



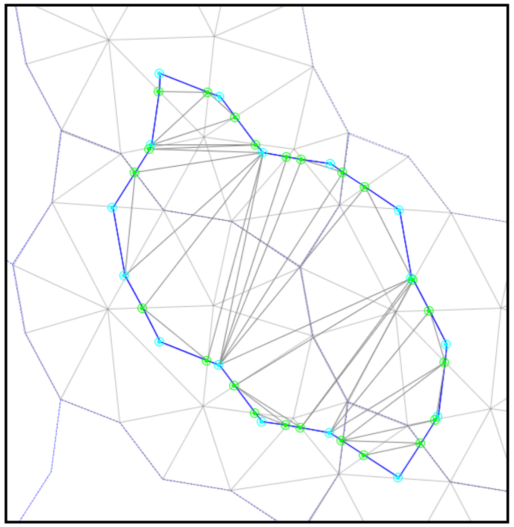
Mesh-Cutting Algorithm (Cont'd)

1. Template offset is specified by user
2. Construct template as an empty boundary "shell" with temporary edges
    - Follow the template to identify intersections with the underlying mesh
    - Load underlying mesh copies on-demand
    - Add fragments that lie inside the template to a queue; check future intersections against the fragment queue

1.



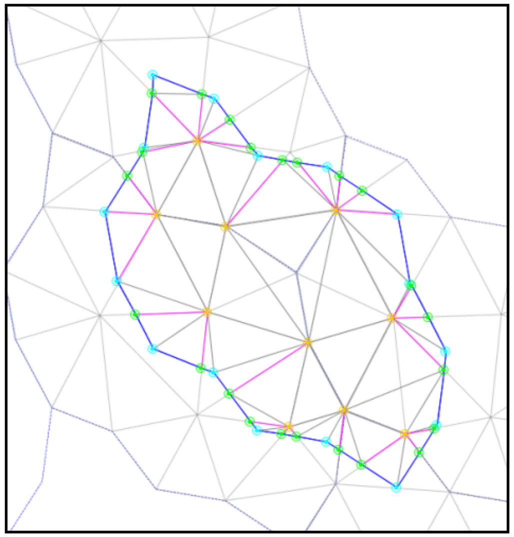
2.



Mesh-Cutting Algorithm (Cont'd)

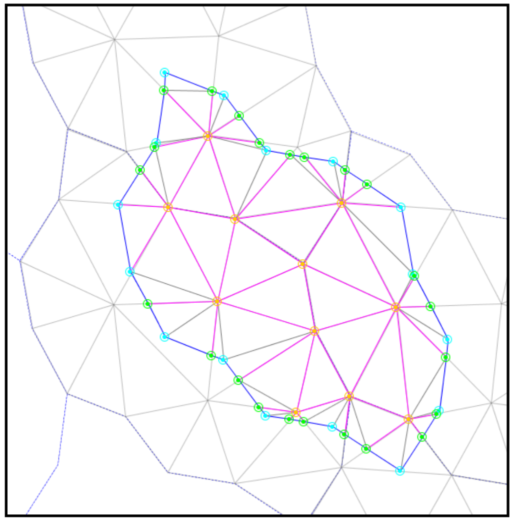
3. After finding all intersections, add fragments as constraint edges to the template shell

3.



4. Perform a flood search to capture the remaining geometry inside the template

4.

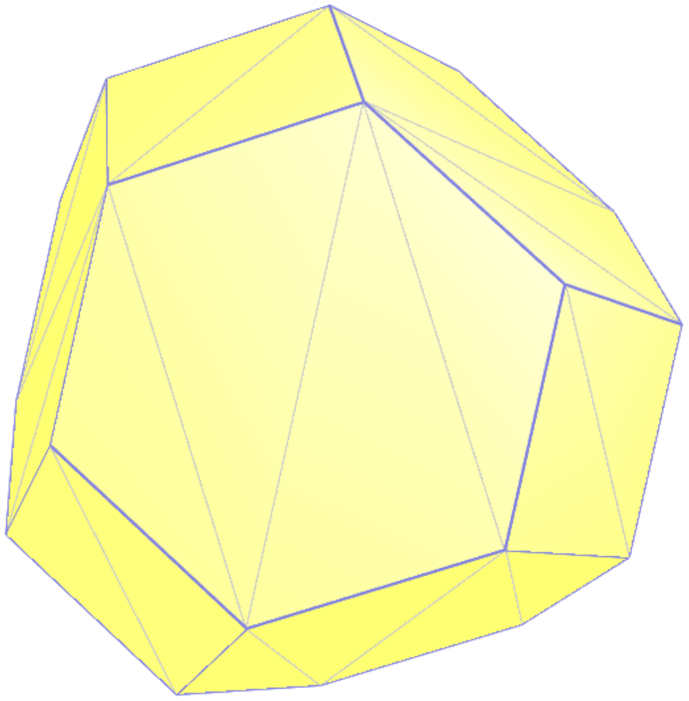


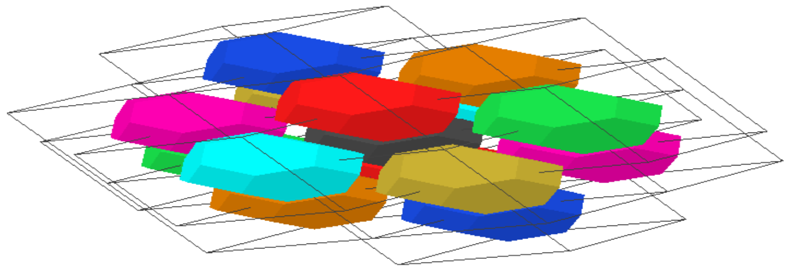
Visual Debugging

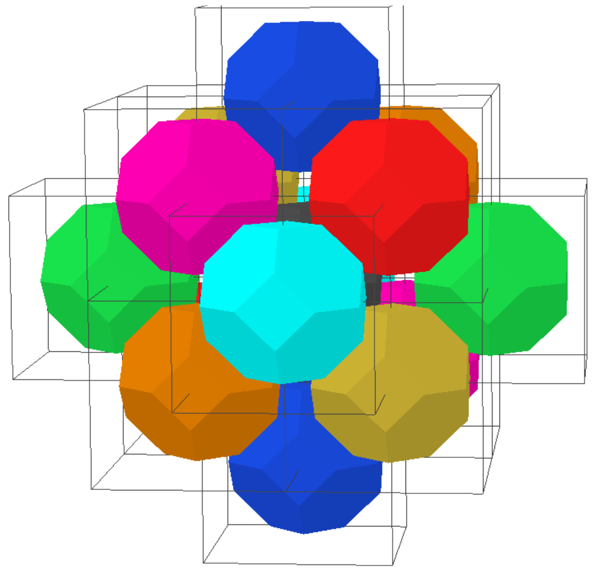
- Sophisticated algorithms are difficult to debug using traditional text-based debuggers/methods
- Geometric/graphics algorithms offer the advantage of visual debugging options
  - Animation and visualization help identify extreme, difficult, or unexpected cases
  - Can isolate and visualize subtasks within the algorithm
- Implemented in Java2D by overriding the event-queue repainting mechanism
  - Events can be inserted mid-algorithm to create visual “breakpoints”
  - Events can specify which geometric features to highlight

3D Tilings

- Exploring two fundamental domains:
  - #1: Truncated octahedron, derived from the body-centered cubic lattice

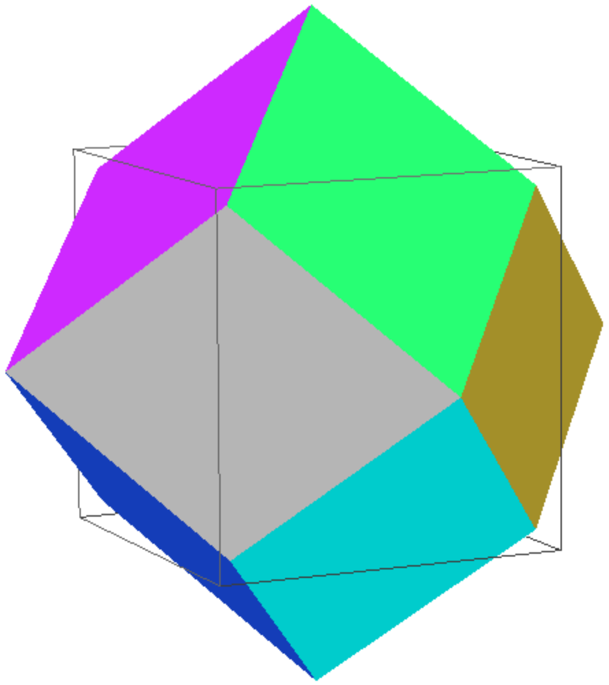
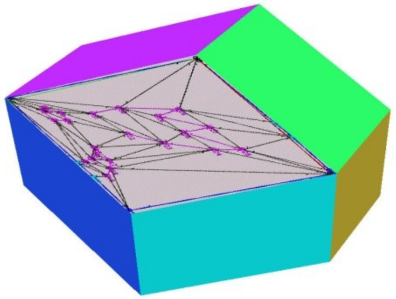
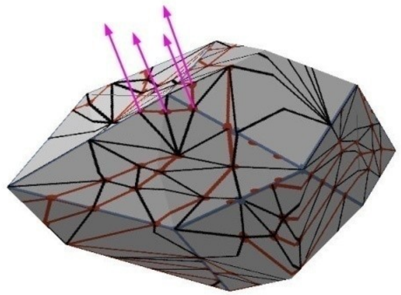
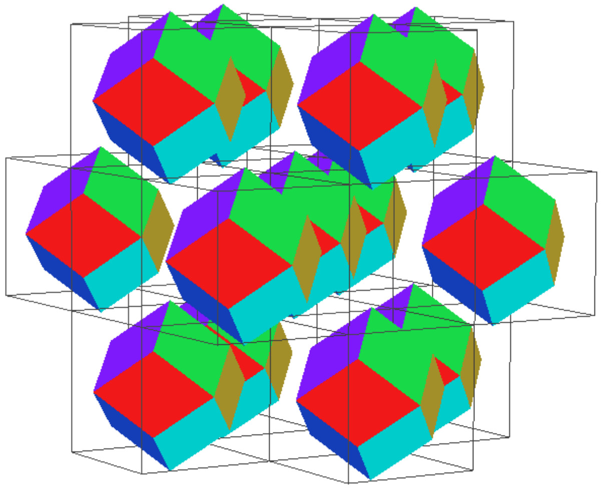






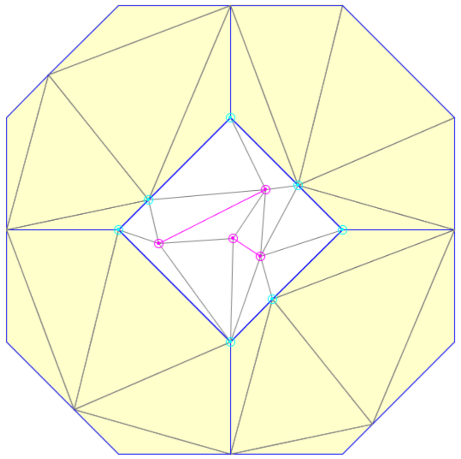
3D Tilings

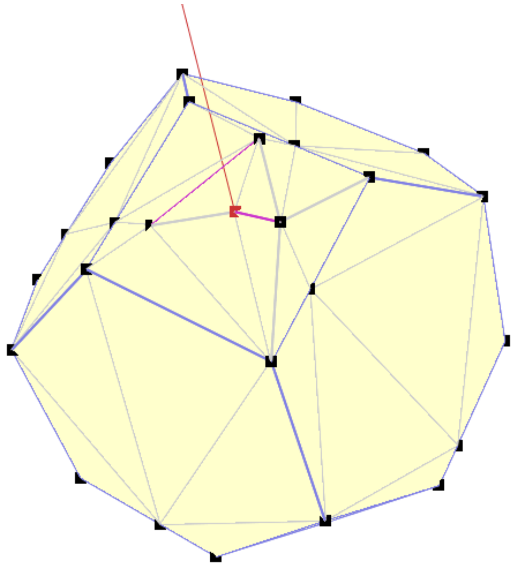
- Exploring two fundamental domains:
  - #2: Rhombic dodecahedron, based on the densest sphere packing

Overview of 3D Editing Interface

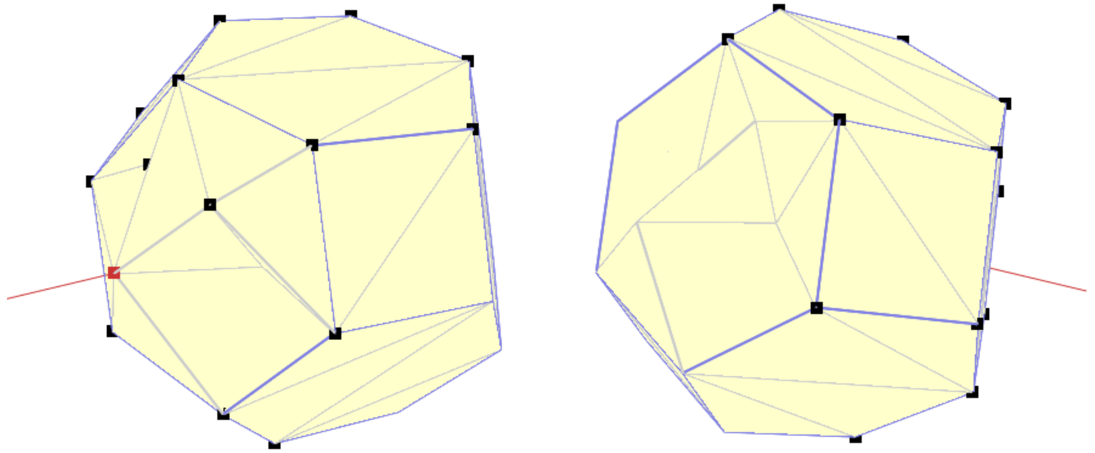
- Phase I:
  - Individual "panes" of the 3D tile can be Delaunay triangulated
  - Vertices can be moved in 2D within pane interior
  - Boundary vertices cannot be moved yet, since this would create non-planar panes
- Phase II:
  - All vertices can be moved in 3D:
    - Last selected point defines an axis through the origin
    - Can move points parallel to axis or in perpendicular plane
  - Local edits available by trisecting faces, but no Delaunay guarantee
  - Limited "roll-back" to Phase I





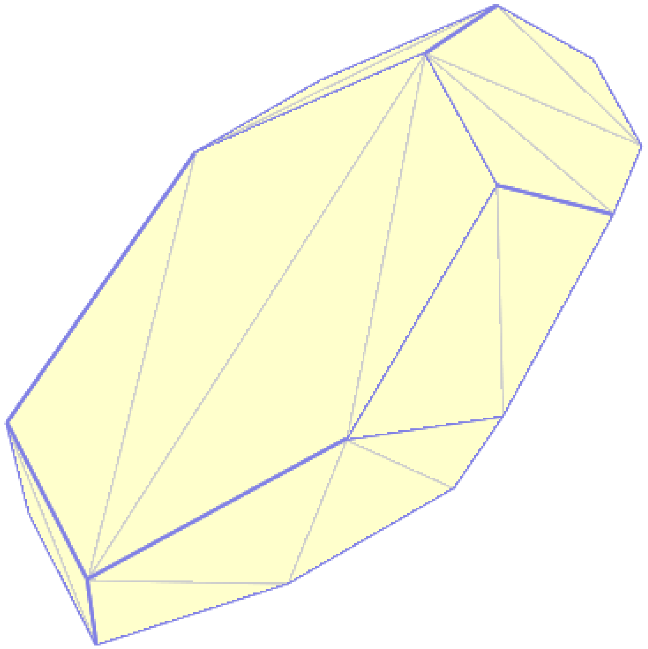
User Interface Issues

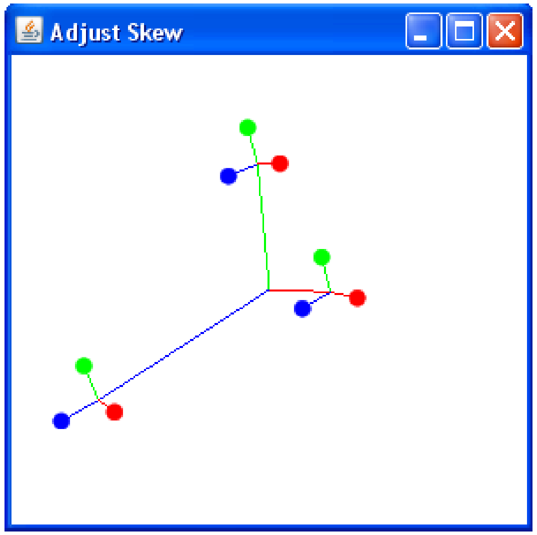
- Occlusion is an obstacle to free-form editing of 3D tiles
  - Because of symmetry, edits in the current view will also change opposite faces
  - Creating a convex feature (e.g. a fish fin) creates a corresponding concave feature (e.g. eye socket) on the opposite side
  - Dual cameras can show convex/concave pairings



User Interface Issues (Cont’d)

- 3D domains can be scaled/skewed and remain space-filling
- What is the easiest way to manipulate this affine transform?
  - Created a widget with 9 control points, each restricted to one degree of freedom
  - Widget maintains same orientation as camera

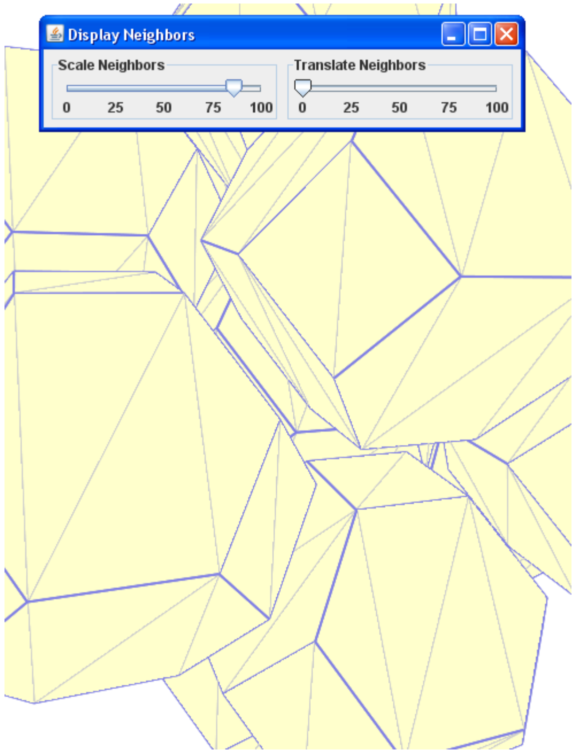


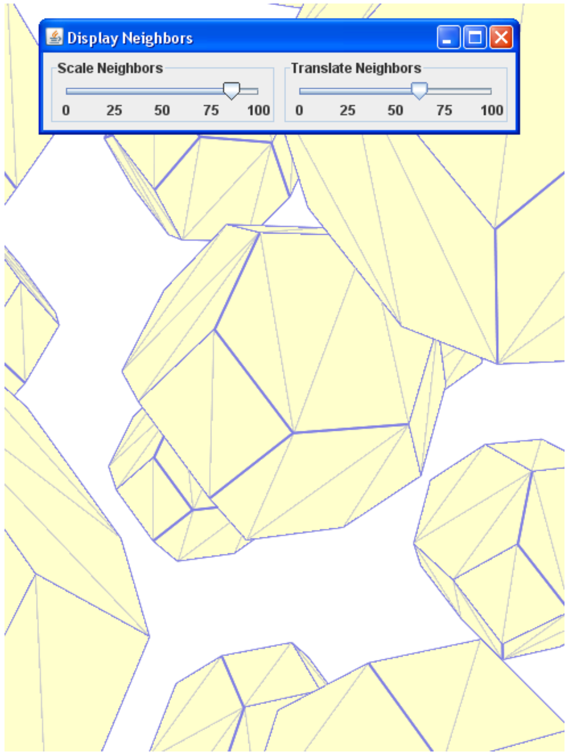


User Interface Issues (Cont'd)

- 3D tilings can have complicated interlocking features
- Nearest neighbors can be scaled/translated to reveal the interface between adjoining tiles

Scaled to 85%





Scaled to 85%
+ translated 0.6
tile lengths away

Conclusion

- Developed a CAD tool for designing layered 2½D tiles
  - Can draw on an existing "vocabulary" of 2D tilings from Escher's sketchbook
- 3D cubic lattice tiles are more difficult to design
  - The entire editable surface is constrained to fit seamlessly with adjacent tiles
    - In the 2D case, only the 1D border is subject to symmetry constraints, while the interior can be decorated freely
  - There is no Escher sketchbook for 3D
- Addressed several UI issues in our prototype 3D tool
  - Two-stage editing allows for Delaunay triangulation
  - Dual cameras reveal occluded features
  - Widget for controlling skew enables high-level editing operations
  - Interactive display of nearest neighbors shows how tiles interlock
- Will address future UI issues as we try to create attractive 3D tiles!